SOEN 422 Lab 1B Discussion

Note:** for this Lab, ***raw C/C++ was interpreted as disallowing higher level functions from the AVR libraries such as *pinMode(), digitalRead(),* etc. Register names however, such as *PORTB*, that are defined within the AVR libraries were permitted.

Task 1

Related file: 40022016_Lab1B_Task1.ino

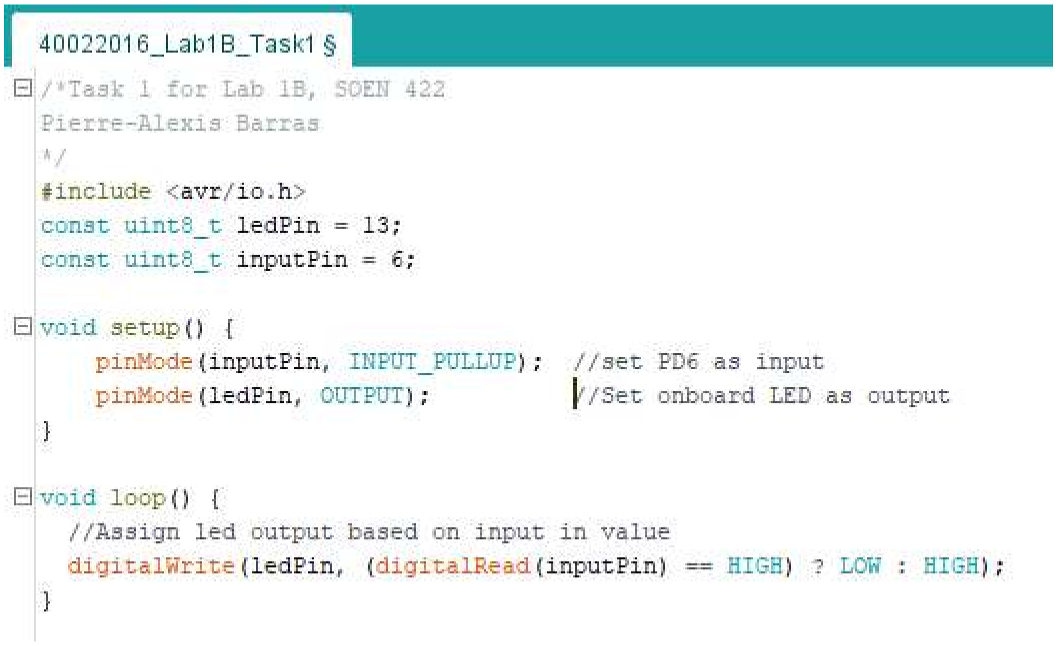
***Note:** Task 1 and Task 2 were interpreted such that the desired output was that when Pin 6 is grounded the LED will turn On, and when Pin 6 is floating/powered the LED shall be OFF.

Implementing the logical inverter was straightforward using the higher-level Arduino commands. Using functions such as *pinMode()* allowed easy setup and manipulation of the registers. uint8_t constants describing the pin locations were used to allow portability.

The process begins by setting all relevant pins via *pinMode()* to the outlined I/O configuration.

The loop constantly assigns a value to the LED via *digitalWrite()* and uses a ternary operator to determine the value by checking the value of the input pin.

The size of this implementation came to 900 bytes (2% of program storage space) with 9 bytes allocated to global variables (~0% of dynamic memory).


40022016_Lab1B_Task1 §

```
/*Task 1 for Lab 1B, SOEN 422
Pierre-Alexis Barras
*/
#include <avr/io.h>
const uint8_t ledPin = 13;
const uint8_t inputPin = 6;

void setup() {
    pinMode(inputPin, INPUT_PULLUP);  //set PD6 as input
    pinMode(ledPin, OUTPUT);          //Set onboard LED as output
}

void loop() {
  //Assign led output based on input in value
  digitalWrite(ledPin, (digitalRead(inputPin) == HIGH) ? LOW : HIGH);
}
```

Task 2

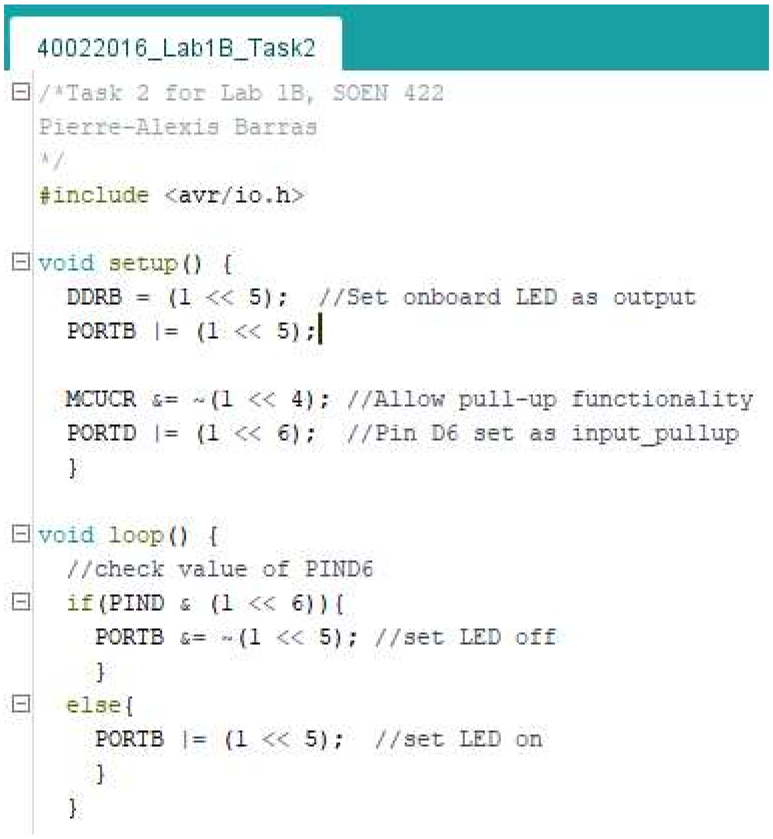
Related file: 40022016_Lab1B_Task2.ino

Implementing the logical inverter using only raw C/C++ proved to be a lot more verbose than in the previous task. Implementing a solution also required the use of the ATmega328P Datasheet to research how to enable input pull-up functionality on the board.

The setup consists of setting the up the onboard LED as an output by toggling the 5th bit in the both the DDRB and PORTB registers. MCUCR's 4th bit toggles the input pullup functionality of the board; it is set to logic one to enable this functionality. PORTD6 was set to logic one to explicitly set pin 6 as an input (even though all pins are set to inputs as a default), and with the MCUCR's 4th bit toggled, outlines it as an input pullup.

The loop consists of an if-statement checking if there is a logic one on pin 6 of the board. If there is, it uses a bitmask to clear the 5th bit of PORTB to turn off the onboard led. Otherwise, it uses a bitmask to set the led output to logic one.

The size of this implementation came to 468 bytes (1% of program storage space) with 9 bytes allocated to global variables (~0% of dynamic memory). This is notable as this implementation is a little over half the size of Task 1 which uses a higher level of implementation.


40022016_Lab1B_Task2

```
/*Task 2 for Lab 1B, SOEN 422
Pierre-Alexis Barras
*/
#include <avr/io.h>

void setup() {
  DDRB = (1 << 5);  //Set onboard LED as output
  PORTB |= (1 << 5);

  MCUCR &= ~(1 << 4); //Allow pull-up functionality
  PORTD |= (1 << 6);  //Pin D6 set as input_pullup
  }

void loop() {
  //check value of PIND6
  if(PIND & (1 << 6)){
    PORTB &= ~(1 << 5); //set LED off
    }
  else{
    PORTB |= (1 << 5);  //set LED on
    }
  }
```

Task 3

Related file: 40022016_Lab1B_Task3.ino

In the implementation of the task, there are two static constant floats used for clarity

- *VREF* - (set to 5.0) refers to the expected reference voltage used on the ATmega328P, for this task, logic 5V.
- *VOLTTHRESHOLD* - (set to 2.0) refers to the minimal voltage (exclusive) that the GP2Y0A21YK distance sensor must produce for the LED to turn on. The value is determined by referring to Figure 5 *Analog Output Voltage vs. Distance to Reflective Object* and drawing a parallel line from the distance axis. Voltages above this line will lie in the range of voltages that will turn on the LED. For this task, a minimum of 4cm is required, which approximates a line at 2V on the diagram. Consequently, this also generates the upper bound of our range, which is estimated to be around 14cm.

To prevent repeated segments of code, a *readAdc(uint8_t)* function was created to safely read the analog info from an analog pin. The function takes a *uint8_t* value to determine which pin it must read from. The function employs a safety mask over the ADMUX register to protect the state of the register's REFSn and ADLAR bits while modifying the MUXn values. Next the ADC control and status register (ADCSRA) is set to start a conversion in single conversion mode. The function then waits for the conversion to finish before return the floating-point value of ADC.

The setup of this implementation consists of firstly explicitly setting the I/O pins to their appropriate setting. The onboard LED is set to an output, and pin PC2 (ADC2) is set as an input.

Next, the ADMUX register is set to 0x40. This value is chosen so that the REFS1 bit is set to 0 and REFS0 bit is set to 1. This value pair is important as it has V_{ref} equal AV_{cc}, or the voltage at the AREF pin for the conversion. Furthermore, the value 0x40 is chosen for AMUX as it will right justify the ADC data register and the value of the MUXn bits are controlled within the *readAdc()* function. The ADSCRA register is then set to that the ADC conversion is enabled and the ADC prescalers is set to 128 so that it operates within the 50-200 kHz ranged outlined in the instructions of the task.

Next, the setup performs a garbage read of the ADC by calling the *readAdc()* function with a parameter of 2 to specify the ADC2 channel pin. The Serial object baud rate is then set to the Arduino IDE default (9600) and a message, signaling the end of the setup function, is sent.

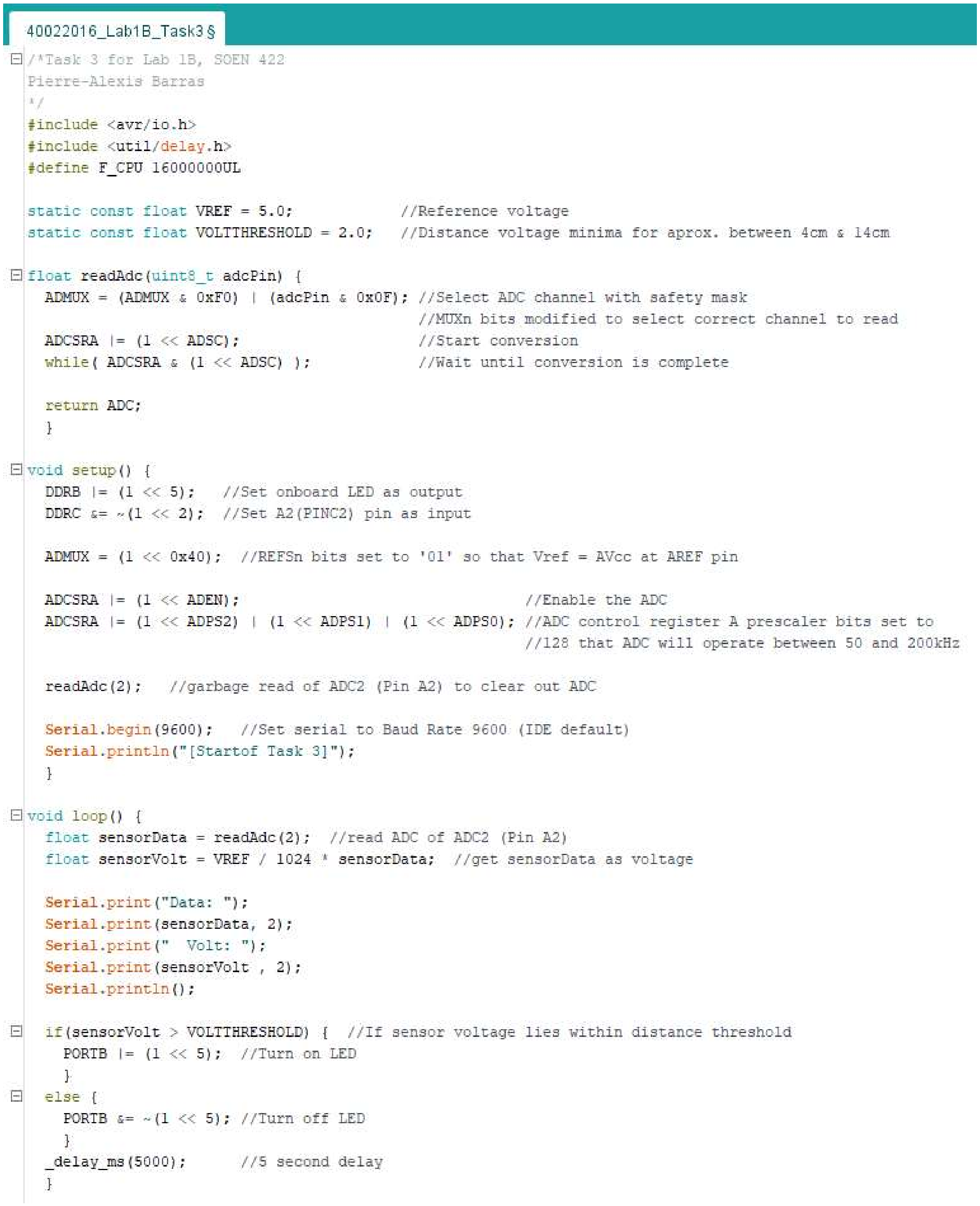
The loop function of the implementation makes use of two floating point local variables: *sensorData* and *sensorVolt*. The former, *sensorData,* is the value read from the ADC conversion on PINC2 obtained from the *readAdc(2)* function. *sensorVolt* is calculated from the formula below:

$$sensorVolt = \frac{VREF * \ sensorData}{1024}$$

Which is derived from the single ended ADC conversion formula found in the ATmega328P datasheet:

$$ADC = \frac{V_{IN} * 1024}{V_{REF}}$$

The loop begins by obtaining the values of *sensorData* and *sensorVolt* and then output them to the serial. Next, *sensorVolt* is compared against the *VOLTTHRESHOLD* value to determine if there is a detected object in the detector’s range. If the *sensorVolt* value is greater than the threshold, the onboard led is turned on, otherwise the LED is turned off. Then, the loop performs a 5s delay courtesy of the *_delay_ms()* function before beginning again. The decision to place the delay at the end of the loop was to ensure that a user would not have to wait 5s at start up for the first valuable read from the detector.



40022016_Lab1B_Task3 §

```
/*Task 3 for Lab 1B, SOEN 422
Pierre-Alexis Barras
*/
#include <avr/io.h>
#include <util/delay.h>
#define F_CPU 16000000UL

static const float VREF = 5.0;           //Reference voltage
static const float VOLTTHRESHOLD = 2.0;   //Distance voltage minima for aprox. between 4cm & 14cm

float readAdc(uint8_t adcPin) {
  ADMUX = (ADMUX & 0xF0) | (adcPin & 0x0F); //Select ADC channel with safety mask
                                            //MUXn bits modified to select correct channel to read
  ADCSRA |= (1 << ADSC);                    //Start conversion
  while( ADCSRA & (1 << ADSC) );            //Wait until conversion is complete

  return ADC;
  }

void setup() {
  DDRB |= (1 << 5);   //Set onboard LED as output
  DDRC &= ~(1 << 2);  //Set A2(PINC2) pin as input

  ADMUX = (1 << 0x40);  //REFSn bits set to '01' so that Vref = AVcc at AREF pin

  ADCSRA |= (1 << ADEN);                                //Enable the ADC
  ADCSRA |= (1 << ADPS2) | (1 << ADPS1) | (1 << ADPS0); //ADC control register A prescaler bits set to
                                                        //128 that ADC will operate between 50 and 200kHz

  readAdc(2);   //garbage read of ADC2 (Pin A2) to clear out ADC

  Serial.begin(9600);   //Set serial to Baud Rate 9600 (IDE default)
  Serial.println("[Startof Task 3]");
  }

void loop() {
  float sensorData = readAdc(2);  //read ADC of ADC2 (Pin A2)
  float sensorVolt = VREF / 1024 * sensorData;  //get sensorData as voltage

  Serial.print("Data: ");
  Serial.print(sensorData, 2);
  Serial.print("  Volt: ");
  Serial.print(sensorVolt , 2);
  Serial.println();

  if(sensorVolt > VOLTTHRESHOLD) {  //If sensor voltage lies within distance threshold
    PORTB |= (1 << 5);  //Turn on LED
    }
  else {
    PORTB &= ~(1 << 5); //Turn off LED
    }
  _delay_ms(5000);      //5 second delay
  }
```

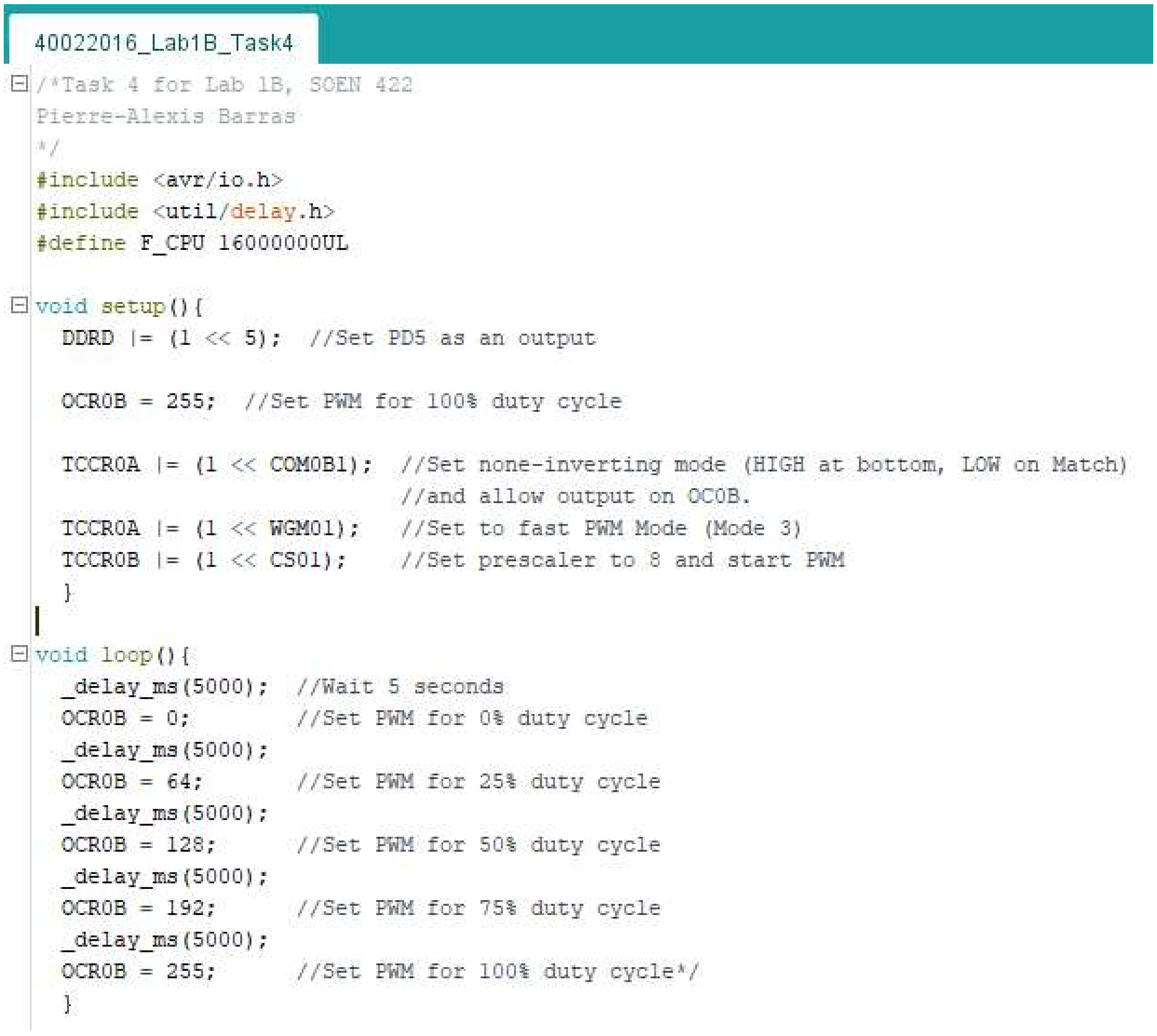
Task 4

Related file: 40022016_Lab1B_Task4.ino

For this task, the *delay.h* library was included to handle the 5 second delay outlined by the ask requirements. The F_CPU was also defined as 16mHz.

For the setup method, the data direction of PIN D5 (OC0B) was set as an output. Next, the output compare 0B register for the pulse width modulation (PWM) was set to 255 or 100% of the duty cycle. This was done to set the external led connected to OC0B to begin as fully on. Next, The COM0B1 bit was set to logic 1 to ensure that the duty cycle could be written to PIN D5. Finally, the timer was set to Mode 3, and then the timer and PWM was started by setting a prescaler of 8.

The loop method consists of a sequence of five second delays from the use of the *_delay_ms()* function followed by setting the output compare register to a predefined value to adjust the LED intensity. The intensities for the external LED follow the following pattern: 0% duty -> 25% duty -> 50% duty ->75% duty -> 100% duty.


40022016_Lab1B_Task4

```
/*Task 4 for Lab 1B, SOEN 422
Pierre-Alexis Barras
*/
#include <avr/io.h>
#include <util/delay.h>
#define F_CPU 16000000UL

void setup(){
  DDRD |= (1 << 5);  //Set PD5 as an output

  OCR0B = 255;  //Set PWM for 100% duty cycle

  TCCR0A |= (1 << COM0B1);  //Set none-inverting mode (HIGH at bottom, LOW on Match)
                            //and allow output on OC0B.
  TCCR0A |= (1 << WGM01);   //Set to fast PWM Mode (Mode 3)
  TCCR0B |= (1 << CS01);    //Set prescaler to 8 and start PWM
  }

void loop(){
  _delay_ms(5000);  //Wait 5 seconds
  OCR0B = 0;        //Set PWM for 0% duty cycle
  _delay_ms(5000);
  OCR0B = 64;       //Set PWM for 25% duty cycle
  _delay_ms(5000);
  OCR0B = 128;      //Set PWM for 50% duty cycle
  _delay_ms(5000);
  OCR0B = 192;      //Set PWM for 75% duty cycle
  _delay_ms(5000);
  OCR0B = 255;      //Set PWM for 100% duty cycle*/
  }
```

Task 5

Related file: 40022016_Lab1B_Task5.ino

For this task, there would need to be 2 interrupts to satisfy the conditions: The first would need to be an external interrupt for when PIND2 (INT0) is pulled low to turn on the LED, the second would be dependent on the timer to turn off the LED after 2.5s of it being turned on.

The setup of the program consists of initially setting the pin statuses for the onboard LED as an output and PIND2 as input pullup. Next, the external interrupt mask register has its 0th bit set to allow external interrupts from INT0. The EICRA register also had the ISC00 bit set cleared to 0 to ensure that the interrupt will occur when there is a logic change on PIND2. This condition of calling the interrupt service routine (ISR) for INT0 was selected over when INT0 is pulled low to ensure/minimizes the amount of interrupt signals produced. It was observed that when EICRA register was set to send an interrupt when INT0 is low, that several sequential interrupts would be sent, leading to inconsistent output (led would remain on while INT0 was grounded).

TIMER1 is employed due to its 16-bit size, as 8-bits would not be large enough to reach a 2.5s threshold with prescalers. The prescaler was selected to be 1024 and OCR1A was deduced to be 0x9895 according to the following formula:

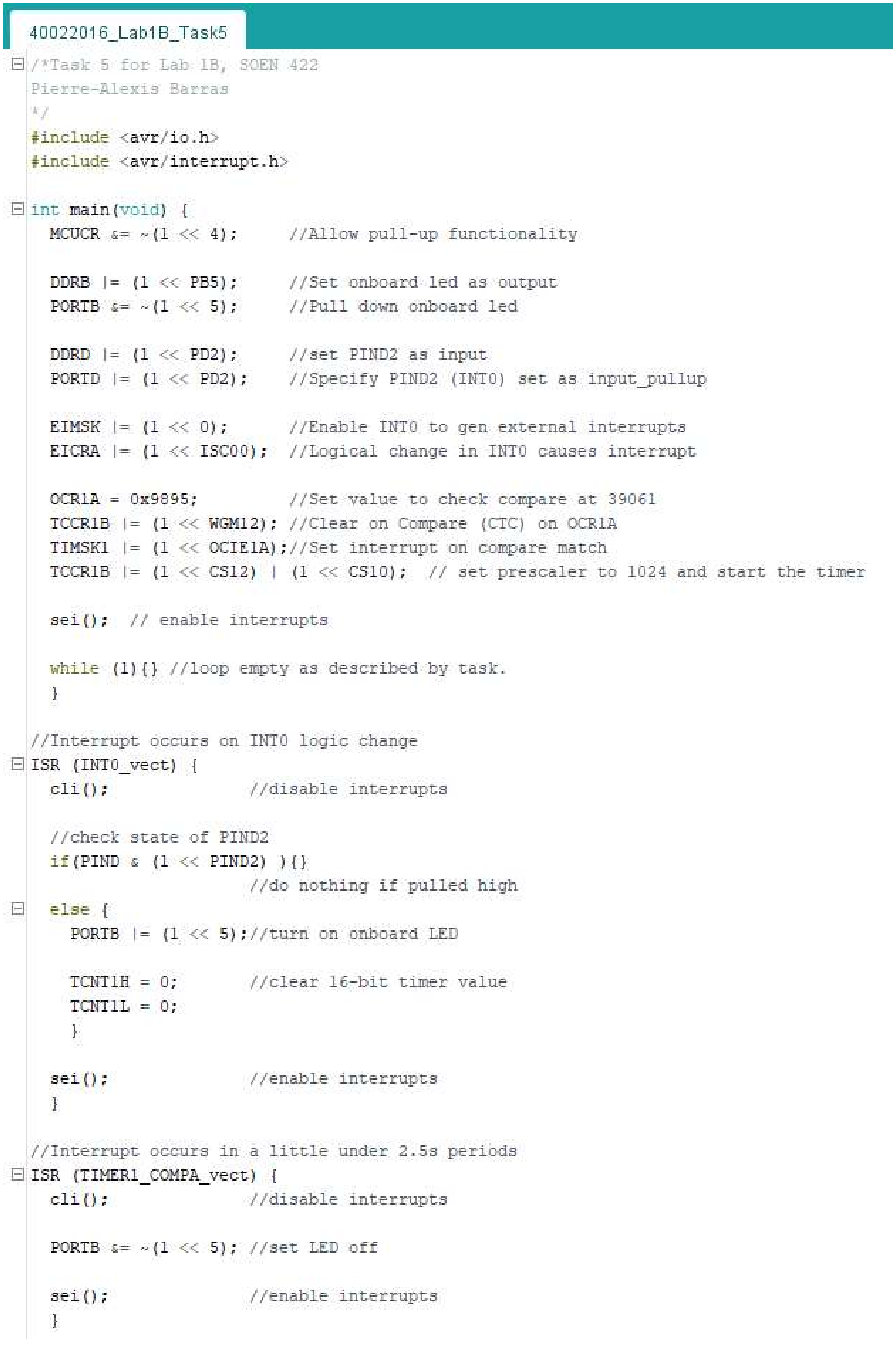
$$
\begin{aligned}
&OCR1A = \left\lfloor \left\lfloor \frac{clock\ speed}{prescaler\ value} * desired\ time \right\rfloor - 1 \right\rfloor \\
&= \left\lfloor \left\lfloor \frac{16000000hz}{1024} * 2.5s \right\rfloor - 1 \right\rfloor \\
&= \lfloor 39062.5 - 1 \rfloor \\
&= 39061 \\
&= 0\mathrm{x}9895
\end{aligned}
$$

TIMER1 is also set to clear on compare to ensure COMPA interrupts every 2.5s. The appropriate timer mask is set so that interrupts are flagged and the timer is cleared on compare and then the timer is started. Finally, interrupts are enabled system via the use of the *sei()* function.

The loop was left blank as outlined by the constraints of this task.

The ISR caught by the INT0 vector occurs when there is a logic change to the INT0 pin (PIND2). The ISR begins by disabling interrupts, and then checking the state of the PIND register. If PIND2 is logic one, nothing occurs and the ISR reenables interrupts and finishes. Else, if PIND2 is pulled low, the onboard LED is lit and the contents of the TIMER1 counter registers (TCNT1H and TCNT1L) are cleared. This is to cause the clock to throw and interrupt after 2.5s of this interrupt. The ISR then reenables interrupts and finishes.

Lastly, the ISR caught by the TIMER1 COMPA vector disables interrupts, turns off the LED and then reenables interrupts.


40022016_Lab1B_Task5

```
/*Task 5 for Lab 1B, SOEN 422
Pierre-Alexis Barras
*/
#include <avr/io.h>
#include <avr/interrupt.h>

int main(void) {
  MCUCR &= ~(1 << 4);     //Allow pull-up functionality

  DDRB |= (1 << PB5);     //Set onboard led as output
  PORTB &= ~(1 << 5);     //Pull down onboard led

  DDRD |= (1 << PD2);     //set PIND2 as input
  PORTD |= (1 << PD2);    //Specify PIND2 (INT0) set as input_pullup

  EIMSK |= (1 << 0);      //Enable INT0 to gen external interrupts
  EICRA |= (1 << ISC00);  //Logical change in INT0 causes interrupt

  OCR1A = 0x9895;         //Set value to check compare at 39061
  TCCR1B |= (1 << WGM12); //Clear on Compare (CTC) on OCR1A
  TIMSK1 |= (1 << OCIE1A);//Set interrupt on compare match
  TCCR1B |= (1 << CS12) | (1 << CS10);  // set prescaler to 1024 and start the timer

  sei();  // enable interrupts

  while (1){} //loop empty as described by task.
  }

//Interrupt occurs on INT0 logic change
ISR (INT0_vect) {
  cli();              //disable interrupts

  //check state of PIND2
  if(PIND & (1 << PIND2) ){}
                      //do nothing if pulled high
  else {
    PORTB |= (1 << 5);//turn on onboard LED

    TCNT1H = 0;       //clear 16-bit timer value
    TCNT1L = 0;
    }

  sei();              //enable interrupts
  }

//Interrupt occurs in a little under 2.5s periods
ISR (TIMER1_COMPA_vect) {
  cli();              //disable interrupts

  PORTB &= ~(1 << 5); //set LED off

  sei();              //enable interrupts
  }
```